Exhibit A

UNITED STATES DISTRICT COURT

NORTHERN DISTRICT OF CALIFORNIA

SAN FRANCISCO DIVISION

ORACLE AMERICA, INC. Plaintiff, v. GOOGLE INC. Defendant.	Case No. CV 10-03561 WHA

OPENING EXPERT REPORT OF JOHN C. MITCHELL REGARDING PATENT INFRINGEMENT

SUBMITTED ON BEHALF OF PLAINTIFF ORACLE AMERICA, INC.

III. EXECUTIVE SUMMARY

8. I have been asked to offer an opinion on whether the following patent claims are infringed by Google:

- Claims 11, 12, 15, 17, 22, 27, 29, 38, 39, 40, and 41 of United States Patent No. RE38,104 ("the '104 patent") (Exhibit A attached to Oracle's infringement contentions submitted to Google on April 1, 2011);
- Claims 1, 2, 3, and 8 of United States Patent No. 6,910,205 ("the '205 patent") (Exhibits B-1 and B-2 attached to Oracle's infringement contentions submitted to Google on April 1, 2011);
- Claims 1, 6, 7, 12, 13, 15, and 16 of United States Patent No. 5,966,702 ("the '702 patent") (Exhibit C attached to Oracle's infringement contentions submitted to Google on April 1, 2011);
- Claims 1, 4, 8, 12, 14, and 20 of United States Patent No. 6,061,520 ("the '520 patent") (Exhibit F attached to Oracle's infringement contentions submitted to Google on April 1, 2011);
- Claims 1, 4, 6, 10, 13, 19, 21, and 22 of United States Patent No. 7,426,720 ("the '720 patent") (Exhibit G attached to Oracle's infringement contentions submitted to Google on April 1, 2011);
- Claims 10 and 11 of United States Patent No. 6,125,447 ("the '447 patent") (Exhibit D attached to Oracle's infringement contentions submitted to Google on April 1, 2011); and
- Claims 13, 14, and 15 of United States Patent No. 6,192,476 ("the '476 patent") (Exhibit E attached to Oracle's infringement contentions submitted to Google on April 1, 2011).

9. Based on my investigation and analysis, it is my opinion that Google, by making, distributing, and using Android, literally meets the limitations of these asserted claims, in the manner described in the Exhibits to Oracle's infringement contentions submitted to Google on April 1, 2011 and in this report.

10. To the extent Android does not literally infringe the claim limitations of the asserted claims of the '205 and '702 patents, Android contains equivalent elements corresponding to each and every requirement of the claim limitations at issue for the reasons detailed below. Therefore, Android meets those limitations literally or under the doctrine of equivalents.

69. Google may have included the Java security framework because it intended to make use of these security features in the future. Google may also have implemented this framework to allow Java developers to port their existing software to the Android platform. Additionally, Google may have wanted to provide developers with the option of using the java.security framework. (*See, e.g.*, RESTful Single Sign on shown on the iPhone, Comment by D. Bornstein, April 14, 2009, *available at* http://groups.google.com/group/android-security-discuss/tree/browse_frm/month/2009-04/03bdf8dd52fe7101?rnum=31&_done=/group/android-security-discuss/browse_frm/month/2009-04?&pli=1 (last visited July 22, 2011) ("We implemented the traditional security framework for applications that wish to perform intra-process security.").)

70. Some of the ways that application developers could benefit from the java.security framework are illustrated by descriptions of the mBS Mobile application. mBS Mobile for Android provides support for Web Widgets. In an Android application that hosts widgets or other components provided by other developers, the application itself should be protected from risks presented by the additional components. This general idea is illustrated by descriptions for the mBS Mobile application. For example, "[t]he OSGi framework is a module system and service platform for the Java programming language that implements a complete and dynamic component model,…" (http://en.wikipedia.org/wiki/OSGi (last visited July 20, 2011).) ProSyst is a company that built software based on the OSGi framework. "ProSyst **mBS Mobile** is a carrier grade embedded OSGi solution integrated and optimized for a variety of smart phone platforms…" (http://www.prosyst.com/index.php/de/html/content/46/Mobile-OSGi-Runtimes/ (last visited July 20, 2011).) Because this application is designed to run additional components that may present security risks, "mBS Mobile is shipped with the Java Security Manager turned on and a default security policy, …" (http://dz.prosyst.com/mbsmobile/android/release_notes/android_func_overview.html (last visited April 26, 2011).) ProSyst distributes an evaluation version of its mBS Mobile OSGi application through the Android Market.

(https://market.android.com/details?id=com.prosyst.mbs.mobile.android&feature=search_result (last visited August 4, 2011).)

71. The java.security framework allows Google Android application developers to implement the "principle of least privilege" within their Java applications. The principle of least privilege is a well-accepted principle for developing secure software, clearly recognized in the Google Android documentation (*e.g.*, http://developer.android.com/guide/topics/fundamentals.html (last visited July 19, 2011).) The principle of least privilege provides each software module with access to *only* the information and resources that are necessary for legitimate purposes. In this way, the computing system may be protected from faults and malicious behavior. Application developers seeking to implement this important security principle within their applications would naturally use the java.security framework. Therefore, Google may have implemented the java.security framework to entice more developers to its platform.

72. Android security via process separation does not provide the same degree of isolation or access control as Java security. The '447 patent (as well as the '476 patent) reflects a fine-grained security model enabled by protection domains associating code with permissions in order to determine whether certain actions may be permitted. Android source code implements the '447 patent (and the '476 patent) security model. Android purports to rely on process separation and operating system level security mechanisms (Linux based), and to that extent already purports to design around Oracle's fine-grained security model. (*See* http://developer.android.com/guide/topics/security/security.html.) However, Android's coarse-grained security model has been criticized as "insufficient protection against third-party applications seeking to collect sensitive data." (*See, e.g.*, W. Enck et al., "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10) at 9, *available at* http://appanalysis.org/tdroid10.pdf (visited Jan. 2011); *see also* D. Goodin, "2 out of 3 Android apps use private data 'suspiciously' Google protections 'insufficient'," posted

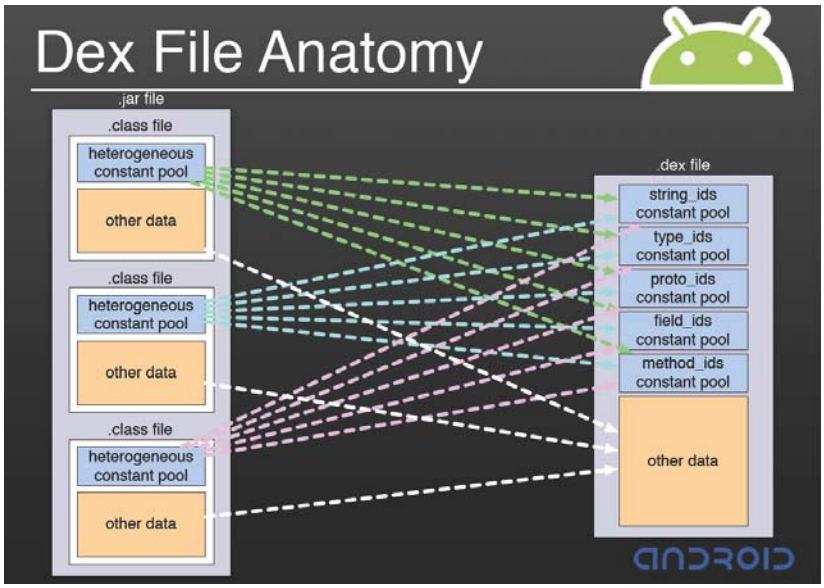
security-discuss/browse_frm/month/2009-04?&pli=1 (last visited July 22, 2011) ("We implemented the traditional security framework for applications that wish to perform intra-process security.").)

77. Some of the ways that application developers could benefit from the java.security framework are illustrated by descriptions of the mBS Mobile application. mBS Mobile for Android provides support for Web Widgets. In an Android application that hosts widgets or other components provided by other developers, the application itself should be protected from risks presented by the additional components. This general idea is illustrated by descriptions for the mBS Mobile application. For example, "[t]he OSGi framework is a module system and service platform for the Java programming language that implements a complete and dynamic component model,…" (http://en.wikipedia.org/wiki/OSGi (last visited July 20, 2011).) ProSyst is a company that built software based on the OSGi framework. "ProSyst **mBS Mobile** is a carrier grade embedded OSGi solution integrated and optimized for a variety of smart phone platforms…" (http://www.prosyst.com/index.php/de/html/content/46/Mobile-OSGi-Runtimes/ (last visited July 20, 2011).) Because this application is designed to run additional components that may present security risks, "mBS Mobile is shipped with the Java Security Manager turned on and a default security policy, …" (http://dz.prosyst.com/mbsmobile/android/release_notes/android_func_overview.html (last visited April 26, 2011).) ProSyst distributes an evaluation version of its mBS Mobile OSGi application through the Android Market. (https://market.android.com/details?id=com.prosyst.mbs.mobile.android&feature=search_result (last visited August 4, 2011).)

78. The java.security framework allows Google Android application developers to implement the "principle of least privilege" within their Java applications. The principle of least privilege is a well-accepted principle for developing secure software, clearly recognized in the Google Android documentation (*e.g.*, http://developer.android.com/guide/topics/fundamentals.html (last visited July 19, 2011).) The

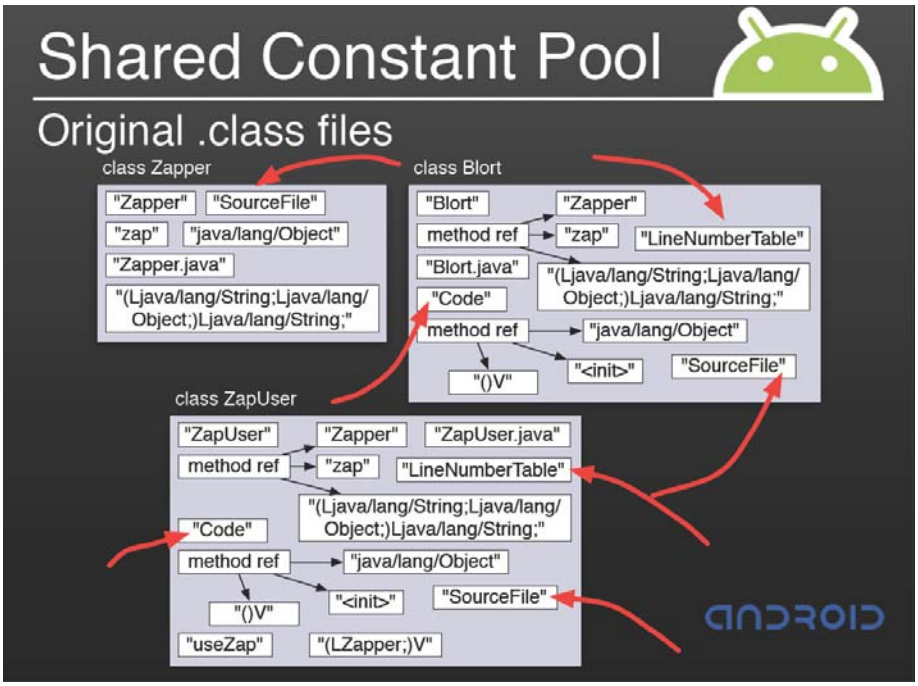
469. The **second element of claim 1**, recites "forming a shared table comprising said plurality of duplicated elements." The Android dx tool forms a shared table of the duplicated elements from the plurality of class files being preprocessed, in the manner discussed above and again here. This process is explained in the Dalvik Video at 7:20–9:25 and Dalvik Presentation, slides 15-20, where the recited shared table includes, *e.g.*, one or more of the "string_ids constant pool," "type_ids constant pool," "proto_ids constant pool," "field_ids constant pool," and "method_ids constant pool."

470. The Dalvik Presentation shows the elements of the class files combining into a shared constant pool (shared tables) in the .dex file.



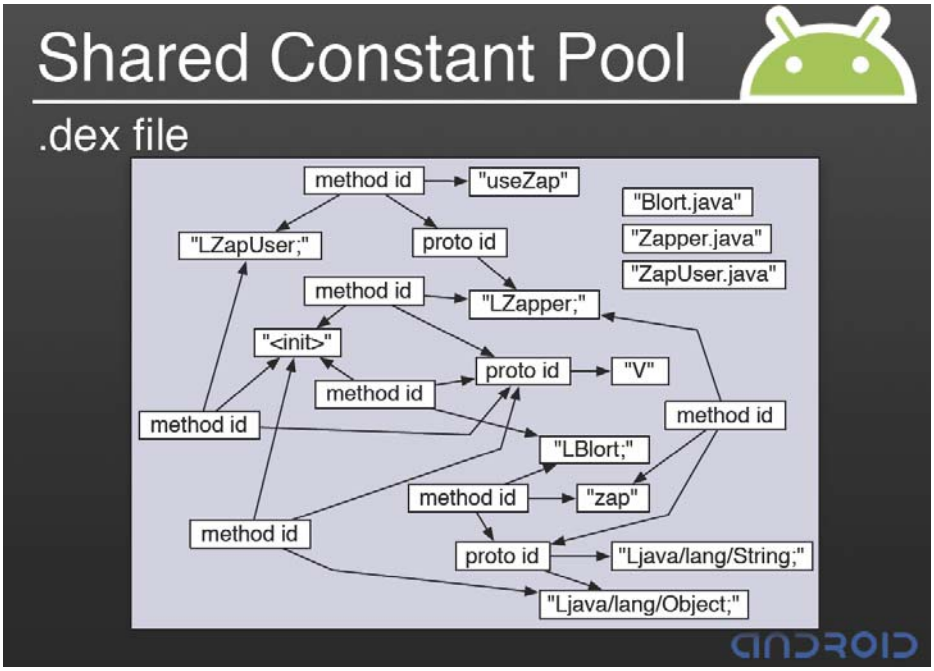
(Dalvik Presentation, Slide 15)

471. In the illustration above, each of "string_ids," "type_ids" and "method_ids" are examples of the shared tables (or, equivalently, a collective shared table). (*See also* 5/16/2011 Bornstein Dep. 91:6-21.) In addition, the discussion of the "Shared Constant Pool" in the Dalvik Video explains that the duplicated elements in the class files are consolidated into the shared constant pool (shared table) of the .dex file. (*See* Dalvik Presentation, Slides 15-21.) For example, slide 19 of the Dalvik Presentation shows the separate class files having duplicated elements.



(Dalvik Presentation, Slide 19)

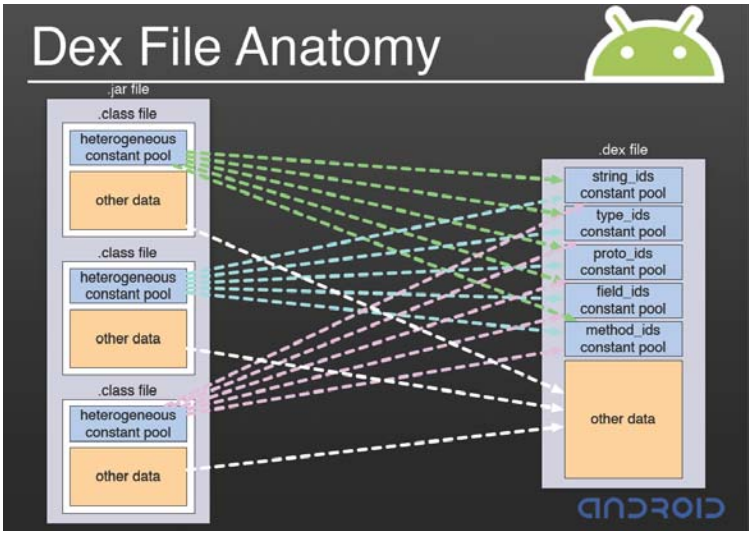
472. Next, slide 20 of the Dalvik Presentation shows a representation of the class files after being processed into a single .dex file, with the duplicate elements removed; the elements are then stored in a shared constant pool (shared table):



(Dalvik Presentation, Slide 20)

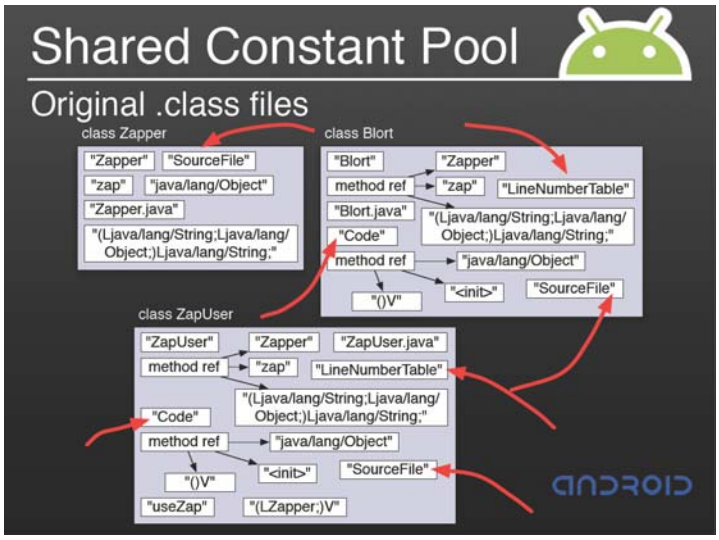
dalvik/dx/src/com/android/dx/dex/file/TypeIdItem.java
dalvik/dx/src/com/android/dx/cf/cst/ConstantPoolParser.java

482. **Claim 5**, which depends from claim 1, recites wherein said step of determining a plurality of duplicated elements comprises "determining one or more constants shared between two or more class files." The Android dx tool determines constants shared between two or more class files. This process is explained in my discussion above and also in the Dalvik Video at 7:20-9:25 and Dalvik Presentation, Slides 11-20. The Dalvik Presentation shows the elements of the class files identified for combining into a shared constant pool (shared tables) in the .dex file.



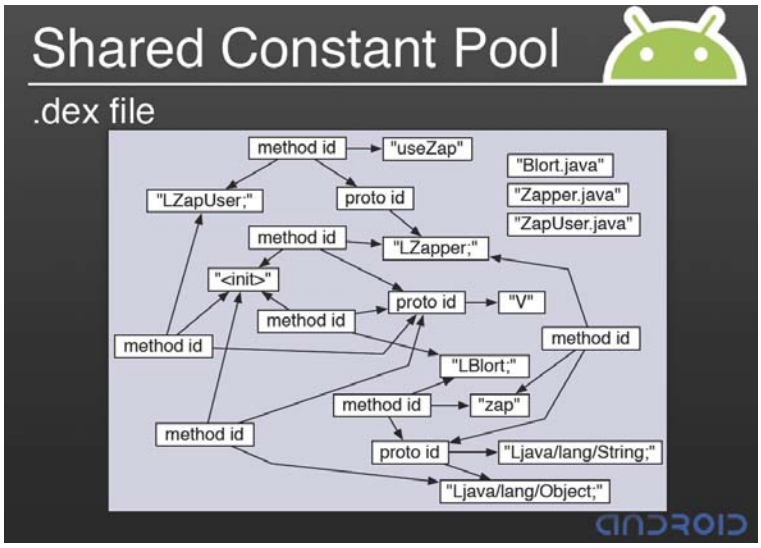
(Dalvik Presentation, Slide 15)

483. In the illustration above, each of "string_ids," "type_ids" and "method_ids" are examples of the shared tables (or, equivalently, a collective shared table). In addition, the discussion of the "Shared Constant Pool" in the Dalvik Video explains that the duplicated elements in the class files are consolidated into the shared constant pool (shared table) of the .dex file. (*See* Dalvik Presentation, Slides 15-21.) For example, slide 19 of the Dalvik Presentation shows the separate class files having duplicated elements.



(Dalvik Presentation, Slide 19)

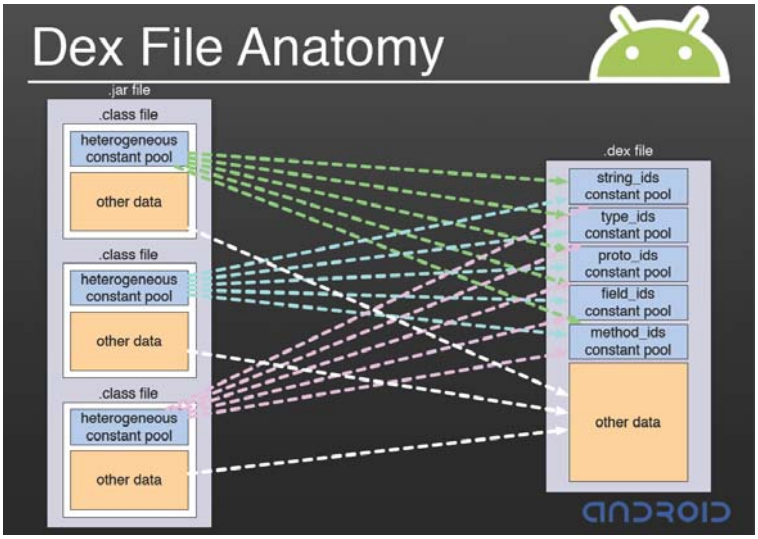
484. Next, slide 20 of the Dalvik Presentation shows a representation of the class files after being processed into a single .dex file, with the duplicate elements removed; the elements are then stored in a shared constant pool (shared table):



(Dalvik Presentation, Slide 20)

485. **Claim 6**, which depends from claim 5, recites wherein said step of forming a shared table comprises "forming a shared constant table comprising said one or more constants shared between said two or more class files." The Android dx tool forms a shared constant table comprising the constants shared between the two or more class files. This process is explained

in my discussion above and also in the Dalvik Video at 7:20–9:25 and Dalvik Presentation, Slide 15. The Dalvik Presentation at 7:20-9:25 shows the elements of the class files combining into a shared constant pool (shared tables) in the .dex file.



(Dalvik Presentation, Slide 15)

486. In the illustration above, each of "string_ids," "type_ids" and "method_ids" are examples of the shared tables (or, equivalently, a collective shared table). In addition, the discussion of the "Shared Constant Pool" in the Dalvik Video at 7:20-9:25 explains that the duplicated elements in the class files are consolidated into the shared constant pool (shared table) of the .dex file. (*See* Dalvik Presentation, Slides 15-21.)

487. **Claim 7** recites a computer program product comprising a computer usable medium having computer readable program code embodied therein for pre processing class files similar to the process described in claim 1. As discussed above, the dx tool, which is part of the Android SDK, is a computer program product and is stored on a tangible and computer useable storage medium, e.g., RAM or hard disk of a computer. It includes computer readable program code for pre-processing class files as recited by the elements of claim 7 as described with respect to claim 1 and detailed in the accompanying charts.

488. **Claim 12**, which depends from claims 11 and 7, recites features similar to those of claims 1, 5, and 6 discussed above. Accordingly, dx tool, stored on a computer usable

494. In the event that the claim were construed in such a way, the formation of a .dex file is at least equivalent to the formation of a multiclass file as required by the claim.

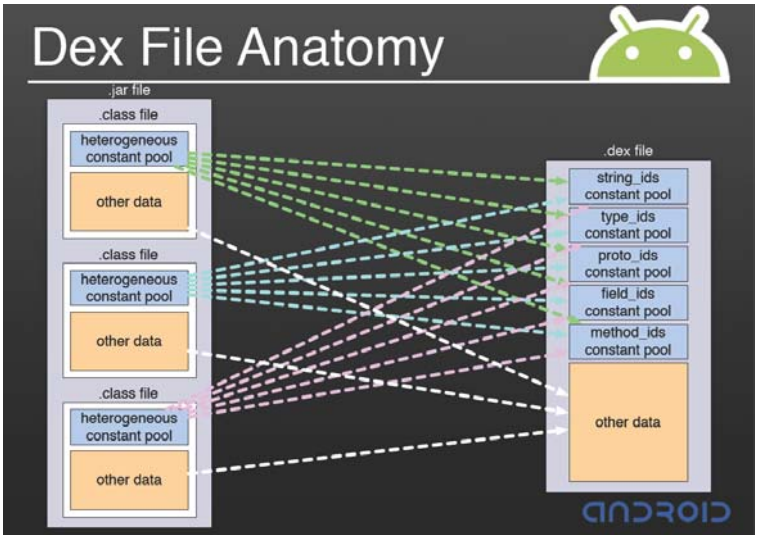
495. To the extent Android does not literally infringe one or more claim elements, Android contains equivalent elements corresponding to each and every requirement of this claim. As described, the dx tool determines duplicated elements in a plurality of class files, forms a shared table comprising said plurality of duplicated elements, removes said duplicated elements from said plurality of class files to obtain a plurality of reduced class files; and forms a multi-class file comprising said plurality of reduced class files and said shared table. Regardless of any possible differences, the dx tool nevertheless (1) performs substantially the same function (to remove duplicated elements of the class files to form a multiclass file, i.e., a .dex file) and (2) works in substantially the same way (to identify duplicated elements of the class files for omission when forming the .dex file) (3) to achieve substantially the same result (a multi class .dex file having reduced class files and a shared table) as required by the claim.

496. Below is a table I have prepared to illustrate that Android, and the dx tool in particular, performs substantially the same function, in substantially the same way, to achieve substantially the same results. The table includes each of the claim elements along with a general description of the function, way, and result associated with each claim element. Additionally, in italics, a description of Androids function, way, and result associated with each claim element is indicated.

Claim Element	**Function**	**Way**	**Result**
determining plurality of duplicated elements in a plurality of class files;	Identify duplicated elements from a set of input class files (or their equivalents). *Android: Identify duplicated elements from a set of input class files.*	Comparison between elements to determine if they are the same. *Android: Use TreeMap data structure to compare individual elements to determine which are duplicates.*	Data structure that represents which elements are determined to be duplicates. *Android: After calls to Intern the Android TreeMap contains one entry for each possibly duplicated element.*

Claim Element	Function	Way	Result
forming a shared table comprising said plurality of duplicated elements;	Form a data structure of duplicated elements. *Android: Form a data structure of duplicated elements.*	Use an internal data structure accessible to the computer program. *Android: Method processClass creates a ClassDefItem object for each class. DexFile.java accumulates the ClassDefItem objects.*	A data structure containing duplicated elements (if present among input class files or their equivalents). *Android: Determination of which elements of input class files are duplicated.*
removing said duplicated elements from said plurality of class files to obtain a plurality of reduced class files; and	Obtain a representation of class files (or their equivalents) with duplicates removed. *Android: Obtain a representation of dex conversion of class files without duplicates.*	Process the class files (or their equivalents) to a form with duplicates removed. A*ndroid: DexFile.add and functions it calls use TreeMap data structures to obtain a representation of class files with duplicates removed.*	A representation of class files (or their equivalents) with duplicates removed. *Android: Data structure containing a representation of dex conversion of multiple class files, with duplicates removed.*
forming a multi-class file comprising said plurality of reduced class files and said shared table.	Produce a single file that represents the content of the input files, but with identified duplicates removed. *Android: The dex tool writes an output .dex file representing multiple class files, with duplicates removed.*	Use data structures representing class file contents and duplicate detection to produce an output file *Android: A .dex file is produced from which duplicates have been removed. Android source code accomplishes this by a series of steps which include creating a ClassDefItem separately for each input Java class without regard for potential duplication of constant pool entries, and once the ClassDefItem is completely formed, it is added to an object of class DexFile by calling DexFile.add(). DexFile.add() calls additional functions*	A single file that represents the content of the input files, but with identified duplicates removed. *Android: The output .dex file represents multiple class files, with duplicates identified by this process removed.*

Claim Element	Function	Way	Result
		that use TreeMap to remove duplicates.	



(Dalvik Presentation, Slide 15)

D. Google Application Source Code

497. I conducted the following experiment to confirm infringement. In particular, I created simple HelloWorld Java programs containing duplicate "Hello World!" strings. I created three duplicates of the "Hello World!" string constant highlighted in the following code:

```
HelloWorld1.java
public class HelloWorld1 {
   public static String test1;

   public void HelloWorld1(){
         test1="Hello World!";
   }
}
```

```
HelloWorld2.java
public class HelloWorld2 {
   public static String test2;

   public void HelloWorld2(){
      test2="Hello World!";
   }
}
```

```
HelloWorldDriver.java
public class HelloWorldDriver {
```

initialization of the array by the preloader." The dx tool steps through the Java .class files, simulating execution of the bytecodes against a memory without executing the byte codes to identify the static initialization of the array by the preloader. For example, Android does not run Java .class files directly; instead, Java .class files are identified and translated into .dex files using the dx utility, where the dx utility simulates the effects of executing the bytecode in order to perform the translation. (*See, e.g.*, dalvik\dx\src\com\android\dx\dex\cf\CfTranslator.java, and dalvik/dx/src/com/android/dx/cf/code/Simulator.java.) These processes identify the static initialization of the array by simulating execution thereof.

523. The Dalvik Video further describes source .class files (slides 41 and 42) for initialization, which includes converting/translating to .dex files (slide 44) and adding elements to a static array to initialize the data (slide 45). (*See, e.g.*, Dalvik Video at 29:50-32:00; Dalvik Presentation, Slides 41-45.)

524. Importantly, in the Android code dalvik/dx/src/com/android/dx/cf/code/Simulator.java, it explicitly provides a "Class which knows how to *simulate* the effects of executing bytecode." (Emphasis added.):

```
/**
 * Class which knows how to simulate the effects of executing bytecode.
 *
 * <p><b>Note:</b> This class is not thread-safe. If multiple threads
 * need to use a single instance, they must synchronize access explicitly
 * between themselves.</p>
 */
public class Simulator {
    /**
     * {@code non-null;} canned error message for local variable
     * table mismatches
     */
    private static final String LOCAL_MISMATCH_ERROR =
        "This is symptomatic of .class transformation tools that ignore " +
        "local variable information.";

    /** {@code non-null;} machine to use when simulating */
    private final Machine machine;

    /** {@code non-null;} array of bytecode */
    private final BytecodeArray code;

    /** {@code non-null;} local variable information */
    private final LocalVariableList localVariables;

    /** {@code non-null;} visitor instance to use */
    private final SimVisitor visitor;

    /**
     * Constructs an instance.
```

```
     *
     * @param machine {@code non-null;} machine to use when simulating
     * @param method {@code non-null;} method data to use
     */
    public Simulator(Machine machine, ConcreteMethod method) {
        if (machine == null) {
            throw new NullPointerException("machine == null");
        }

        if (method == null) {
            throw new NullPointerException("method == null");
        }

        this.machine = machine;
        this.code = method.getCode();
        this.localVariables = method.getLocalVariables();
        this.visitor = new SimVisitor();
    }

    /**
     * Simulates the effect of executing the given basic block. This modifies
     * the passed-in frame to represent the end result.
     *
     * @param bb {@code non-null;} the basic block
     * @param frame {@code non-null;} frame to operate on
     */
    public void simulate(ByteBlock bb, Frame frame) {
        int end = bb.getEnd();

        visitor.setFrame(frame);

        try {
            for (int off = bb.getStart(); off < end; /*off*/) {
                int length = code.parseInstruction(off, visitor);
                visitor.setPreviousOffset(off);
                off += length;
            }
        } catch (SimException ex) {
            frame.annotate(ex);
            throw ex;
        }
    }

    /**
     * Simulates the effect of the instruction at the given offset, by
     * making appropriate calls on the given frame.
     *
     * @param offset {@code >= 0;} offset of the instruction to simulate
     * @param frame {@code non-null;} frame to operate on
     * @return the length of the instruction, in bytes
     */
    public int simulate(int offset, Frame frame) {
        visitor.setFrame(frame);
        return code.parseInstruction(offset, visitor);
    }

    /**
     * Constructs an "illegal top-of-stack" exception, for the stack
     * manipulation opcodes.
     */
    private static SimException illegalTos() {
        return new SimException("stack mismatch: illegal " +
                "top-of-stack for opcode");
    }

    /**
     * Bytecode visitor used during simulation.
     */
    private class SimVisitor implements BytecodeArray.Visitor {
        /**
```

```
     * {@code non-null;} machine instance to use (just to avoid excessive
     * cross-object field access)
     */
    private final Machine machine;

    /**
     * {@code null-ok;} frame to use; set with each call to
     * {@link Simulator#simulate}
     */
    private Frame frame;

    /** offset of the previous bytecode */
    private int previousOffset;

    /**
     * Constructs an instance.
     */
    public SimVisitor() {
        this.machine = Simulator.this.machine;
        this.frame = null;
    }
```

The dx code can simulate the effect of running bytecode that initializes an array, which follows the newarray bytecode. (*See*, *e.g.*, dalvik/dx/src/com/android/dx/cf/code/BytecodeArray.java):

```
    /**
     * Helper to deal with {@code newarray}.
     *
     * @param offset the offset to the {@code newarray} opcode itself
     * @param visitor {@code non-null;} visitor to use
     * @return instruction length, in bytes
     */
    private int parseNewarray(int offset, Visitor visitor) {
        int value = bytes.getUnsignedByte(offset + 1);
        CstType type;
        switch (value) {
            case ByteOps.NEWARRAY_BOOLEAN: {
                type = CstType.BOOLEAN_ARRAY;
                break;
            }
            case ByteOps.NEWARRAY_CHAR: {
                type = CstType.CHAR_ARRAY;
                break;
            }
            case ByteOps.NEWARRAY_DOUBLE: {
                type = CstType.DOUBLE_ARRAY;
                break;
            }
            case ByteOps.NEWARRAY_FLOAT: {
                type = CstType.FLOAT_ARRAY;
                break;
            }
            case ByteOps.NEWARRAY_BYTE: {
                type = CstType.BYTE_ARRAY;
                break;
            }
            case ByteOps.NEWARRAY_SHORT: {
                type = CstType.SHORT_ARRAY;
                break;
            }
            case ByteOps.NEWARRAY_INT: {
                type = CstType.INT_ARRAY;
                break;
            }
            case ByteOps.NEWARRAY_LONG: {
                type = CstType.LONG_ARRAY;
                break;
```

```
        }
        default: {
            throw new SimException("bad newarray code " +
                    Hex.u1(value));
        }
    }

    // Revisit the previous bytecode to find out the length of the array
    int previousOffset = visitor.getPreviousOffset();
    ConstantParserVisitor constantVisitor = new ConstantParserVisitor();
    int arrayLength = 0;

    /*
     * For visitors that don't record the previous offset, -1 will be
     * seen here
     */
    if (previousOffset >= 0) {
        parseInstruction(previousOffset, constantVisitor);
        if (constantVisitor.cst instanceof CstInteger &&
                constantVisitor.length + previousOffset == offset) {
            arrayLength = constantVisitor.value;

        }
    }

    /*
     * Try to match the array initialization idiom. For example, if the
     * subsequent code is initializing an int array, we are expecting the
     * following pattern repeatedly:
     *  dup
     *  push index
     *  push value
     *  *astore
     *
     * where the index value will be incrimented sequentially from 0 up.
     */
    int nInit = 0;
    int curOffset = offset+2;
    int lastOffset = curOffset;
    ArrayList<Constant> initVals = new ArrayList<Constant>();

    if (arrayLength != 0) {
        while (true) {
            boolean punt = false;

            // First check if the next bytecode is dup
            int nextByte = bytes.getUnsignedByte(curOffset++);
            if (nextByte != ByteOps.DUP)
                break;

            // Next check if the expected array index is pushed to the stack
            parseInstruction(curOffset, constantVisitor);
            if (constantVisitor.length == 0 ||
                    !(constantVisitor.cst instanceof CstInteger) ||
                    constantVisitor.value != nInit)
                break;

            // Next, fetch the init value and record it
            curOffset += constantVisitor.length;

            // Next find out what kind of constant is pushed onto the stack
            parseInstruction(curOffset, constantVisitor);
            if (constantVisitor.length == 0 ||
                    !(constantVisitor.cst instanceof CstLiteralBits))
                break;

            curOffset += constantVisitor.length;
            initVals.add(constantVisitor.cst);

            nextByte = bytes.getUnsignedByte(curOffset++);
```

```
                    // Now, check if the value is stored to the array properly
                    switch (value) {
                        case ByteOps.NEWARRAY_BYTE:
                        case ByteOps.NEWARRAY_BOOLEAN: {
                            if (nextByte != ByteOps.BASTORE) {
                                punt = true;
                            }
                            break;
                        }
                        case ByteOps.NEWARRAY_CHAR: {
                            if (nextByte != ByteOps.CASTORE) {
                                punt = true;
                            }
                            break;
                        }
                        case ByteOps.NEWARRAY_DOUBLE: {
                            if (nextByte != ByteOps.DASTORE) {
                                punt = true;
                            }
                            break;
                        }
                        case ByteOps.NEWARRAY_FLOAT: {
                            if (nextByte != ByteOps.FASTORE) {
                                punt = true;
                            }
                            break;
                        }
                        case ByteOps.NEWARRAY_SHORT: {
                            if (nextByte != ByteOps.SASTORE) {
                                punt = true;
                            }
                            break;
                        }
                        case ByteOps.NEWARRAY_INT: {
                            if (nextByte != ByteOps.IASTORE) {
                                punt = true;
                            }
                            break;
                        }
                        case ByteOps.NEWARRAY_LONG: {
                            if (nextByte != ByteOps.LASTORE) {
                                punt = true;
                            }
                            break;
                        }
                        default:
                            punt = true;
                            break;
                    }
                    if (punt) {
                        break;
                    }
                    lastOffset = curOffset;
                    nInit++;
                }
            }

            /*
             * For singleton arrays it is still more economical to
             * generate the aput.
             */
            if (nInit < 2 || nInit != arrayLength) {
                visitor.visitNewarray(offset, 2, type, null);
                return 2;
            } else {
                visitor.visitNewarray(offset, lastOffset - offset, type, initVals);
                return lastOffset - offset;
            }
        }
```

(*See also* 5/16/2011 Bornstein Dep. 205:12-217:9.)

525. The **fourth element of claim 1** recites "storing into an output file an instruction requesting the static initialization of the array." The dx tool stores an instruction requesting the static initialization of the array. This process is described, for example, in the Dalvik Video at 29:50-32:00 and Dalvik Presentation slides 41-45. This is further found in the Android code at Main.java in the dexer package \dalvik\dx\src\com\android\dx\command\dexer:

```
/**
 * Converts {@link #outputDex} into a {@code byte[]}, write
 * it out to the proper file (if any), and also do whatever human-oriented
 * dumping is required.
 *
 * @return {@code null-ok;} the converted {@code byte[]} or {@code null}
 * if there was a problem
 */
private static byte[] writeDex() {
    byte[] outArray = null;

    try {
        OutputStream out = null;
        OutputStream humanOutRaw = null;
        OutputStreamWriter humanOut = null;
        try {
            if (args.humanOutName != null) {
                humanOutRaw = openOutput(args.humanOutName);
                humanOut = new OutputStreamWriter(humanOutRaw);
            }

            if (args.methodToDump != null) {
                /*
                 * Simply dump the requested method. Note: The call
                 * to toDex() is required just to get the underlying
                 * structures ready.
                 */
                outputDex.toDex(null, false);
                dumpMethod(outputDex, args.methodToDump, humanOut);
            } else {
                /*
                 * This is the usual case: Create an output .dex file,
                 * and write it, dump it, etc.
                 */
                outArray = outputDex.toDex(humanOut, args.verboseDump);

                if ((args.outName != null) && !args.jarOutput) {
                    out = openOutput(args.outName);
                    out.write(outArray);
                }
            }

            if (args.statistics) {
                DxConsole.out.println(outputDex.getStatistics().toHuman());
            }
        } finally {
            if (humanOut != null) {
                humanOut.flush();
            }
            closeOutput(out);
            closeOutput(humanOutRaw);
        }
    } catch (Exception ex) {
```

```
                     * to have non-empty successors and yet not have a
                     * primary successor.
                     */
                    primarySuccessorIndex = -1;
                } else {
                    primarySuccessorIndex = catches.size();
                }
            } else {
                insn = new PlainInsn(rop, pos, dest, sources);
            }

            insns.add(insn);

            if (moveResult != null) {
                insns.add(moveResult);
            }

            /*
             * If initValues is non-null, it means that the parser has
             * seen a group of compatible constant initialization
             * bytecodes that are applied to the current newarray. The
             * action we take here is to convert these initialization
             * bytecodes into a single fill-array-data ROP which lays out
             * all the constant values in a table.
             */
            if (initValues != null) {
                extraBlockCount++;
                insn = new FillArrayDataInsn(Rops.FILL_ARRAY_DATA, pos,
                        RegisterSpecList.make(moveResult.getResult()), initValues,
                        cst);
                insns.add(insn);
            }
        }
```

(dalvik/dx/src/com/android/dx/cf/code/RopperMachine.java.)

526. Classes.dex is a usual output of the dx tool. In the skeletonapp example, dx creates classes.dex, in which the instructions requesting the static initialization of the two arrays has been stored:

```
000400:                                        |[000400]
com.example.android.skeletonapp.skeletonapp.<clinit>:()V
000410: 1241                                   |0000: const/4 v1, #int 4 // #4
000412: 2310 1200                              |0001: new-array v0, v1, [I // type@0012
000416: 2600 0d00 0000                         |0003: fill-array-data v0, 00000010 //
+0000000d
00041c: 6900 0700                              |0006: sput-object v0,
Lcom/example/android/skeletonapp/skeletonapp;.tester:[I // field@0007
000420: 2310 1200                              |0008: new-array v0, v1, [I // type@0012
000424: 2600 1200 0000                         |000a: fill-array-data v0, 0000001c //
+00000012
00042a: 6900 0400                              |000d: sput-object v0,
Lcom/example/android/skeletonapp/skeletonapp;.bigger:[I // field@0004
00042e: 0e00                                   |000f: return-void
000430: 0003 0400 0400 0000 0100 0000 0200 ... |0010: array-data (12 units)
000448: 0003 0400 0400 0000 ff00 0000 7f00 ... |001c: array-data (12 units)
```

527. The **fifth, and final, element of claim 1** recites "interpreting the instruction by a virtual machine to perform the static initialization of the array." The Dalvik virtual machine

```
|[1a4] SimpleArrayTest.<init>:()V
|[1bc] SimpleArrayTest.main:([Ljava/lang/String;)V
|
|statistics:
|  class data: 1 item; 22 bytes total
|    22 bytes/item
|  class def: 1 item; 32 bytes total
|    32 bytes/item
|  code: 3 items; 140 bytes total
|    24..84 bytes/item; average 46
|  debug info: 3 items; 17 bytes total
|    5..7 bytes/item; average 5
|  field id: 2 items; 16 bytes total
|    8 bytes/item
|  header: 1 item; 112 bytes total
|    112 bytes/item
|  map list: 1 item; 160 bytes total
|    160 bytes/item
|  method id: 5 items; 40 bytes total
|    8 bytes/item
|  proto id: 3 items; 36 bytes total
|    12 bytes/item
|  string data: 17 items; 228 bytes total
|    3..28 bytes/item; average 13
|  string id: 17 items; 68 bytes total
|    4 bytes/item
|  type id: 8 items; 32 bytes total
|    4 bytes/item
|  type list: 2 items; 12 bytes total
|    6 bytes/item
```

541. This .dex file shows that the lengthy bytecode instructions are replaced by a single fill-array-data Dalvik instruction accompanied by a simple listing of the integer data values that are to be placed in the array.

542. I have confirmed through an illustrative, simple experiment that the dx tool determines the operation corresponding to static initialization of array of primitive data type. Because the dx tool contains an algorithm to play execute bytecode to determine its operation and to create reduced number of instructions to perform operation (here the fill-array-data instruction), the dx tool will similarly play execute and create reduced instructions for other Google application source code (e.g., Gmail, Maps, Google Services Framework, Google Calendar, CertInstaller, Contacts, Desk Clock, Email, Google Quick Search Box, Launcher2, Google Partner Setup, Vending, and MarketUpdater).

543. By grepping and searching the dexdump files corresponding to .odex files for Google applications found on, for example, the Nexus S (*e.g.*, Gmail, Maps, Google Services

605. The copy-on-write in the Linux fork executed by Android is described as follows in Robert Lowe, *Linux Kernel Process Management* (April 15, 2005), a sample chapter is *available at* http://www.informit.com/articles/article.aspx?p=370047&seqNum=2&rll=1.

"Copy-on-Write
…In Linux, fork() is implemented through the use of copy-on-write pages. Copy-on-write (or COW) is a technique to delay or altogether prevent copying of the data. Rather than duplicate the process address space, the parent and the child can share a single copy. The data, however, is marked in such a way that if it is written to, a duplicate is made and each process receives a unique copy."

606. The Linux fork executed by Android provides the "copy-on-write process cloning mechanism" in its *fork()* system call. Linux provides additional "process cloning mechanisms" in its *vfork()* and *clone()* system calls. These Android Linux process cloning mechanisms are described as follows at http://book.opensourceproject.org.cn/kernel/kernelpri/opensource/0131181637/ch03lev1sec3.html#ch03fig09.

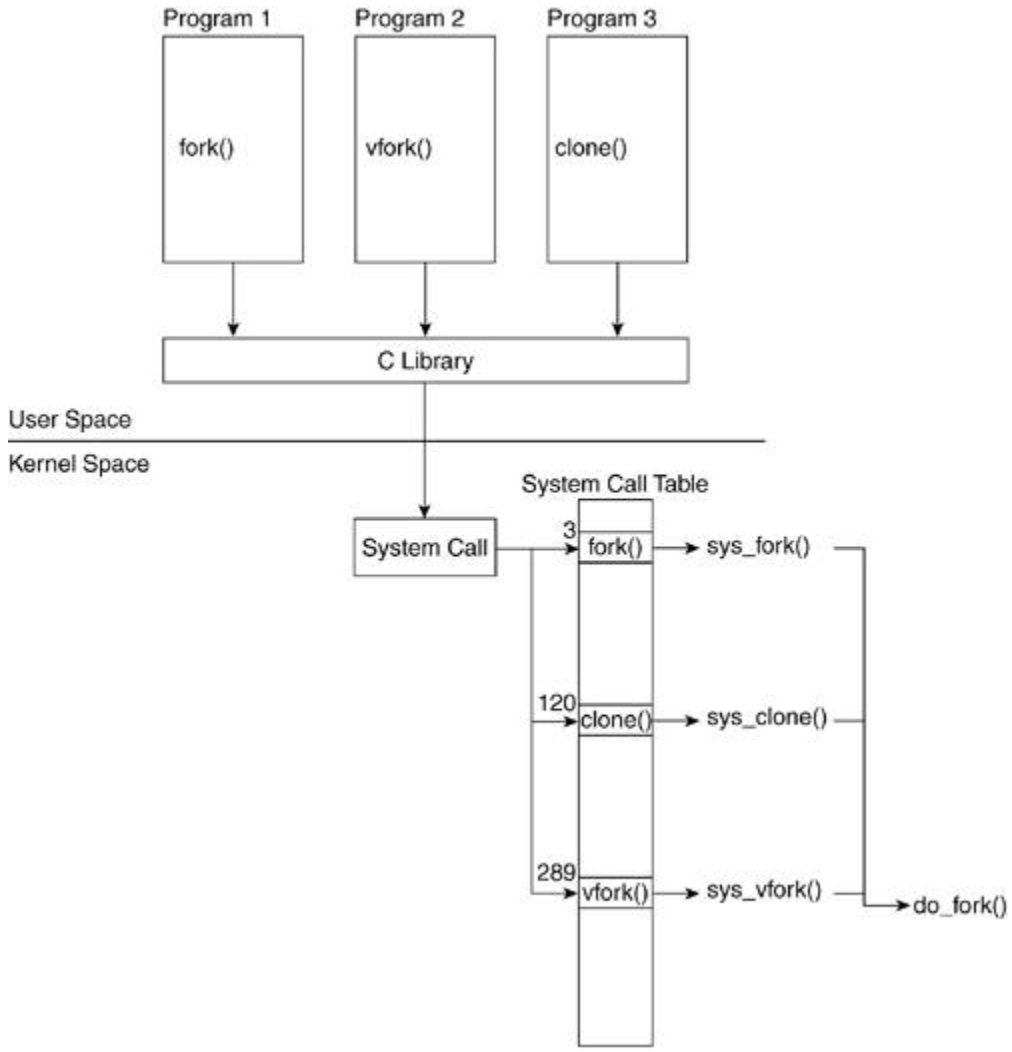
3.3. Process Creation: fork(), vfork(), and clone() System Calls

After the sample code is compiled into a file (in our case, an ELF executable[2]), we call it from the command line. Look at what happens when we press the Return key. We already mentioned that any given process is created by another process. The operating system provides the functionality to do this by means of the `fork()`, `vfork()`, and `clone()` system calls.

[2] ELF executable is an executable format that Linux supports. Chapter 9 discusses the ELF executable format.

The C library provides three functions that issue these three system calls. The prototypes of these functions are declared in `<unistd.h>`. Figure 3.9 shows how a process that calls `fork()` executes the system call `sys_fork()`. This figure describes how kernel code performs the actual process creation. In a similar manner, `vfork()` calls `sys_fork()`, and `clone()` calls `sys_clone()`.

Figure 3.9. Process Creation System Calls



All three of these system calls eventually call `do_fork()`, which is a kernel function that performs the bulk of the actions related to process creation. You might wonder why three different functions are available to create a process. Each function slightly differs in how it creates a process, and there are specific reasons why one would be chosen over the other.

When we press Return at the `shell` prompt, the shell creates the new process that executes our program by means of a call to `fork()`. In fact, if we type the command `ls` at the shell and press Return, the pseudocode of the shell at that moment looks something like this:

```
if( (pid = fork()) == 0 )
   execve("foo");
else
   waitpid(pid);
```

We can now look at the functions and trace them down to the system call. Although our program calls `fork()`, it could just as easily have called `vfork()` or `clone()`, which is why we introduced all three functions in this section. The first function we look at is `fork()`. We delve through the

calls `fork()`, `sys_fork()`, and `do_fork()`. We follow that with `vfork()` and finally look at `clone()` and trace them down to the `do_fork()` call.

3.3.1. fork() Function

The `fork()` function returns twice: once in the parent and once in the child process. If it returns in the child process, `fork()` returns 0. If it returns in the parent, `fork()` returns the child's PID. When the `fork()` function is called, the function places the necessary information in the appropriate registers, including the index into the system call table where the pointer to the system call resides. The processor we are running on determines the registers into which this information is placed.

At this point, if you want to continue the sequential ordering of events, look at the "Interrupts" section in this chapter to see how `sys_fork()` is called. However, it is not necessary to understand how a new process gets created.

Let's now look at the `sys_fork()` function. This function does little else than call the `do_fork()` function. Notice that the `sys_fork()` function is architecture dependent because it accesses function parameters passed in through the system registers.

```
-----------------------------------------------------------------------
arch/i386/kernel/process.c
asmlinkage int sys_fork(struct pt_regs regs)
{
   return do_fork(SIGCHLD, regs.esp, &regs, 0, NULL, NULL);
}
-----------------------------------------------------------------------

-----------------------------------------------------------------------
arch/ppc/kernel/process.c
int sys_fork(int p1, int p2, int p3, int p4, int p5, int p6,
             struct pt_regs *regs)
{
        CHECK_FULL_REGS(regs);
        return do_fork(SIGCHLD, regs->gpr[1], regs, 0, NULL, NULL);
}
-----------------------------------------------------------------------
```

The two architectures take in different parameters to the system call. The structure `pt_regs` holds information such as the stack pointer. The fact that `gpr[1]` holds the stack pointer in PPC, whereas `%esp`[3] holds the stack pointer in x86, is known by convention.

[3] Recall that in code produced in "AT&T" format, registers are prefixed with a %.

3.3.2. vfork() Function

The `vfork()` function is similar to the `fork()` function with the exception that the parent process is blocked until the child calls `exit()` or `exec()`.

`sys_vfork()`

```
arch/i386/kernel/process.c
asmlinkage int sys_vfork(struct pt_regs regs)
{
   return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, regs.ep, &regs, 0, NULL,
NULL);
}
-----------------------------------------------------------------------
arch/ppc/kernel/process.c
int sys_vfork(int p1, int p2, int p3, int p4, int p5, int p6,
              struct pt_regs *regs)
{
        CHECK_FULL_REGS(regs);
        return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, regs->gpr[1],
                       regs, 0, NULL, NULL);
}
-----------------------------------------------------------------------
```

The only difference between the calls to `sys_fork()` in `sys_vfork()` and `sys_fork()` are the flags that `do_fork()` is passed. The presence of these flags are used later to determine if the added behavior just described (of blocking the parent) will be executed.

3.3.3. clone() Function

The `clone()` library function, unlike `fork()` and `vfork()`, takes in a pointer to a function along with its argument. The child process created by `do_fork()`calls this function as soon as it gets created.

```
[View full width]
-----------------------------------------------------------------------
sys_clone()
arch/i386/kernel/process.c
asmlinkage int sys_clone(struct pt_regs regs)
{
 unsigned long clone_flags;
        unsigned long newsp;
        int __user *parent_tidptr, *child_tidptr;

        clone_flags = regs.ebx;
        newsp = regs.ecx;
        parent_tidptr = (int __user *)regs.edx;
        child_tidptr = (int __user *)regs.edi;
        if (!newsp)
                newsp = regs.esp;
        return do_fork(clone_flags & ~CLONE_IDLETASK, newsp, &regs, 0,
parent_tidptr,
 child_tidptr);
}
-----------------------------------------------------------------------

-----------------------------------------------------------------------
arch/ppc/kernel/process.c
int sys_clone(unsigned long clone_flags, unsigned long usp,
              int __user *parent_tidp, void __user *child_thread\
```

```
ptr,
              int __user *child_tidp, int p6,
              struct pt_regs *regs)
{
        CHECK_FULL_REGS(regs);
        if (usp == 0)
                usp = regs->gpr[1];     /* stack pointer for chi\
ld */
        return do_fork(clone_flags & ~CLONE_IDLETASK, usp, regs,\
 0,
                        parent_tidp, child_tidp);
}
```

As Table 3.4 shows, the only difference between `fork()`, `vfork()`, and `clone()` is which flags are set in the subsequent calls to `do_fork()`.

Table 3.4. Flags Passed to do_fork by fork(), vfork(), and clone()

	fork()	vfork()	clone()
SIGCHLD	X	X	
CLONE_VFORK		X	
CLONE_VM		X	

Finally, we get to `do_fork()`, which performs the real process creation. Recall that up to this point, we only have the parent executing the call to `fork()`, which then enables the system call `sys_fork()`; we still do not have a new process. Our program `foo` still exists as an executable file on disk. It is not running or in memory.

3.3.4. do_fork() Function

We follow the kernel side execution of `do_fork()` line by line as we describe the details behind the creation of a new process.

[View full width]

```
kernel/fork.c
1167   long do_fork(unsigned long clone_flags,
1168       unsigned long stack_start,
1169       struct pt_regs *regs,
1170       unsigned long stack_size,
1171       int __user *parent_tidptr,
1172       int __user *child_tidptr)
1173   {
1174     struct task_struct *p;
```

```
1175     int trace = 0;
1176     long pid;
1177
1178     if (unlikely(current->ptrace)) {
1179        trace = fork_traceflag (clone_flags);
1180        if (trace)
1181          clone_flags |= CLONE_PTRACE;
1182     }
1183
1184   p = copy_process(clone_flags, stack_start, regs, stack_size,
parent_tidptr,
 child_tidptr);
-----------------------------------------------------------------------
```

Lines 1178-1183

The code begins by verifying if the parent wants the new process `ptraced`. ptracing references are prevalent within functions dealing with processes. This book explains only the `ptrace` references at a high level. To determine whether a child can be traced, `fork_traceflag()` must verify the value of `clone_flags`. If `CLONE_VFORK` is set in `clone_flags`, if `SIGCHLD` is not to be caught by the parent, or if the current process also has `PT_TRACE_FORK` set, the child is traced, unless the `CLONE_UNTRACED` or `CLONE_IDLETASK` flags have also been set.

Line 1184

This line is where a new process is created and where the values in the registers are copied out. The `copy_process()` function performs the bulk of the new process space creation and descriptor field definition. However, the start of the new process does not take place until later. The details of `copy_process()` make more sense when the explanation is scheduler-centric. See the "Keeping Track of Processes: Basic Scheduler Construction" section in this chapter for more detail on what happens here.

607. Example source code files that provide the copy-on-write process cloning mechanism include the following.

libcore\dalvik\src\main\java\dalvik\system\Zygote.java,
dalvik\vm\native\dalvik_system_Zygote.c,
linux-2.6\arch\arv32\kernel\process.c,
linux-2.6\kernel\fork.c.

608. Note that the above process.c file is for particular system architecture and is only an example. Other architectures have similar process.c files.

609. An example call chain to various code modules that provide the copy-on-write process cloning mechanism include the following.

forkAndSpecialize calls forkAndSpecializeCommon,
forkAndSpecializeCommon calls fork,
fork call sys_fork,
sys_fork calls do_fork,

```
        if (!okay) {
            clazz->status = CLASS_ERROR;
            if (!dvmCheckException(dvmThreadSelf())) {
                dvmThrowException("Ljava/lang/VirtualMachineError;", NULL);
            }
        }
        if (interfaceIdxArray != NULL) {
            free(interfaceIdxArray);
        }
        return okay;
    }
```

620. **Claim 6**, which depends from claim 1, recites a "process cloning mechanism to instantiate the child runtime system process by copying the memory space of the master runtime system process into a separate memory space for the child runtime system process."

621. The process cloning mechanism is described as follows at http://developer.android.com/guide/basics/what-is-android.html.

"Android Runtime
…The Dalvik VM relies on the Linux kernel for underlying functionality such as threading and low-level memory management.

Linux Kernel
Android relies on Linux version 2.6 for core system services such as security, memory management, process management, network stack, and driver model. The kernel also acts as an abstraction layer between the hardware and the rest of the software stack."

622. Example source code files that provide the process cloning mechanism, e.g., the *clone()* system call, include the following.

linux-2.6\arch\arv32\kernel\process.c,
linux-2.6\kernel\fork.c.

Note that the above process.c file is for particular system architecture and is only an example. Other architectures have similar process.c files.

623. An example call chain to various code modules that provide the clone() process cloning mechanism include the following.

clone calls sys_clone,
sys_clone calls do_fork,
do_fork calls copy_process.

624. The file linux-2.6\arch\arv32\kernel\process.c provides a *sys_clone()* system call, invoked by a *clone()* system call, that can provide the "process cloning mechanism" by not setting a CLONE_VM flag to pass by the argument *clone_flags* in the *do_fork()* module.

```
asmlinkage int sys_clone(unsigned long clone_flags, unsigned long newsp,
      void __user *parent_tidptr, void __user *child_tidptr,
      struct pt_regs *regs)
{
   if (!newsp)
      newsp = regs->sp;
   return do_fork(clone_flags, newsp, regs, 0, parent_tidptr,
         child_tidptr);
}
```

625. The website http://linux.die.net/man/2/clone describes the CLONE_VM flag as follows.

> **CLONE_VM**
>
> If CLONE_VM is set, the calling process and the child processes run in the same memory space. In particular, memory writes performed by the calling process or by the child process are also visible in the other process. Moreover, any memory mapping or unmapping performed with *mmap(2)* or *munmap(2)* by the child or calling process also affects the other process.
>
> If CLONE_VM is not set, the child process runs in a separate copy of the memory space of the calling process at the time of clone(). Memory writes or file mappings/unmappings performed by one of the processes do not affect the other, as with *fork(2)*.

626. The *clone()* system call is also described in the excerpt above from the website http://book.opensourceproject.org.cn/kernel/kernelpri/opensource/0131181637/ch03lev1sec3.html#ch03fig09.

627. The file linux-2.6\kernel\fork.c provides the fork code *do_fork()* to perform the process cloning "to instantiate the child runtime system process by copying the memory space of the master runtime system process into a separate memory space for the child runtime system process." The unset CLONE_VM flag from the *clone()* system call can be passed as an argument for parameter *clone_flags* to *do_fork()*, which passes it to *copy_process()*, which performs the process cloning. As stated previously, because *clone()* does not set the CLONE_VM flag, *do_fork()* provides the "process cloning mechanism" of claim 6 "to

instantiate the child runtime system process by copying the memory space of the master runtime system process into a separate memory space for the child runtime system process."

```
/*
 *  Ok, this is the main fork-routine.
 *
 * It copies the process, and if successful kick-starts
 * it and waits for it to finish using the VM if required.
 */
long do-fork(unsigned long clone-flags,
              unsigned long stack-start,
              struct pt_regs *regs,
              unsigned long stack-size,
              int __user *parent_tidptr,
              int __user *child_tidptr)
{
        struct task_struct *p;
        int trace = 0;
        long nr;
…
        p = copy-process(clone-flags, stack-start, regs, stack-size,
                        wake_up_new_task(p, clone-flags);
…
        Tracehook-report-clone-complete(trace, regs,
                                  Clone-flags, nr, p);
…
        return nr;
}
```

628. **Claim 10** recites a "method for dynamic preloading of classes through memory space cloning of a master runtime system process." The method recites features similar to those of claim 1 discussed above. Accordingly, Android and the Android SDK execute the *zygote* process, which performs dynamic preloading of classes through memory space cloning of a master runtime system process, according to the steps of the claim, for the reasons discussed above.

629. **Claim 13**, which depends from claim 10, recites "resolving the class definition." This step recites features similar to those of claim 4 discussed above. Accordingly, the *zygote* process performs this step.

630. **Claim 19** recites a "computer-readable storage medium holding code for performing the method according to claim 10." A computer loaded with Android or the Android SDK will comprise code stored (or held) on a computer-readable storage medium that includes the *zygote* code that performs the method of claim 10, as discussed above.

631. **Claim 21**, which depends from claim 1, recites a "resource controller to set operating system level resource management parameters on the child runtime system process."

```
             * stuff, like where it comes from for bootstrap classes).
             */
            if (clazz != NULL) {
                //LOGI("SETTING pd '%s' to %p\n", clazz->descriptor, pd);
                dvmSetFieldObject((Object*) clazz, gDvm.offJavaLangClass_pd, pd);
            }

            free(descriptor);
            RETURN_PTR(clazz);
        }
```

678. **Claim 10, element C** recites the step of "determining whether an action requested by a particular object is permitted based on said association between said one or more protection domains and said one or more classes." Android meets the conditions of this element as shown in Android's ProtectionDomain.java, Policy.java, SecurityManager.java, AccessController.java, and AccessControlContext.java. The SecurityManager provides security verification facilities for applications. The SecurityManager utilizes the structures provided by the other security classes (listed above) to determine whether an action requested by a particular object is permitted based on the association between protection domains and classes.

679. The Android code below shows Google's method of determining whether an action is permitted based on the association between protection domains and classes. First, the SecurityManager "[c]hecks whether [a] calling thread is allowed to access the resource being guarded by the specific permission object": that is, SecurityManager checks whether the action is permitted. (See comment before checkPermission in SecurityManager.java.) (*See, e.g.*, Android APIs for "java.lang.SecurityManager," *available at* http://developer.android.com/reference/java/lang/SecurityManager.html ("SecurityManager contains a set of checkXXX methods which determine if it is safe to perform a specific operation such as establishing network connections, modifying files, and many more.").)

680. The SecurityManager performs the check by calling AccessController.checkPermission to check the permission.

```
In Froyo, dalvik\libcore\luni\src\main\java\java\lang\SecurityManager.java
In Gingerbread, libcore\luni\src\main\java\java\lang\SecurityManager.java:
/**
 * <strong>Warning:</strong> security managers do <strong>not</strong> provide a
 * secure environment for executing untrusted code. Untrusted code cannot be
 * safely isolated within the Dalvik VM.
 *
 * <p>Provides security verification facilities for applications. {@code
```

```
 * SecurityManager} contains a set of {@code checkXXX} methods which determine
 * if it is safe to perform a specific operation such as establishing network
 * connections, modifying files, and many more. In general, these methods simply
 * return if they allow the application to perform the operation; if an
 * operation is not allowed, then they throw a {@link SecurityException}. The
 * only exception is {@link #checkTopLevelWindow(Object)}, which returns a
 * boolean to indicate permission.
 */
public class SecurityManager {
…
    /**
     * Checks whether the calling thread is allowed to access the resource being
     * guarded by the specified permission object.
     *
     * @param permission
     *             the permission to check.
     * @throws SecurityException
     *              if the requested {@code permission} is denied according to
     *              the current security policy.
     */
    public void checkPermission(Permission permission) {
        try {
            inCheck = true;
            AccessController.checkPermission(permission);
        } finally {
            inCheck = false;
        }
    }

    /**
     * Checks whether the specified security context is allowed to access the
     * resource being guarded by the specified permission object.
     *
     * @param permission
     *             the permission to check.
     * @param context
     *             the security context for which to check permission.
     * @throws SecurityException
     *              if {@code context} is not an instance of {@code
     *              AccessControlContext} or if the requested {@code permission}
     *              is denied for {@code context} according to the current
     *              security policy.
     */
    public void checkPermission(Permission permission, Object context) {
        try {
            inCheck = true;
            // Must be an AccessControlContext. If we don't check
            // this, then applications could pass in an arbitrary
            // object which circumvents the security check.
            if (context instanceof AccessControlContext) {
                ((AccessControlContext) context).checkPermission(permission);
            } else {
                throw new SecurityException();
            }
        } finally {
            inCheck = false;
        }
    }
}
```

681. The AccessController.checkPermission method checks the permission by calling the AccessControlContext.checkPermission method. According to Google, a "permission is considered granted if every ProtectionDomain in the current execution context has been granted the specific permission." (*See* comment before checkPermission in AccessController.java.)

732. Other methods of detecting when a request for an action is made by a principal may be found in Android. Android's implementation of the Java core libraries is replete with code having the form:

```
/**
        SecurityManager currentManager = System.getSecurityManager();
        if (currentManager != null) {
            currentManager.checkPermission([some permission]);
        }

See, e.g. ObjectOutputStream.java.
```

733. Security code such as this means that code that is invoked when principal requests to perform an action will proceed to invoke the code that implements the remaining steps of the claim.

734. **Claim 10, element B** recites the step of "in response to detecting the request, determining whether said action is authorized based on permissions associated with a plurality of routines in a calling hierarchy associated with said principal." Google's Android includes SecurityManager.java to determine whether an action is authorized. (*See, e.g.*, Android APIs for "java.lang.SecurityManager," *available at* http://developer.android.com/reference/java/lang/SecurityManager.html ("Provides security verification facilities for applications.").)

735. The Android code below shows Google's method of determining whether an action is authorized based on permissions associated with a plurality of routines in a calling hierarchy, in response to detecting a request. The SecurityManager "checks whether the calling thread is allowed to access the resource being guarded by the specified permission." (*See* comment before checkPermission in SecurityManager.java.)

```
In Froyo, dalvik\libcore\luni\src\main\java\java\lang\SecurityManager.java
In Gingerbread, libcore\luni\src\main\java\java\lang\SecurityManager.java:
/**
 * <strong>Warning:</strong> security managers do <strong>not</strong> provide a
 * secure environment for executing untrusted code. Untrusted code cannot be
 * safely isolated within the Dalvik VM.
 *
 * <p>Provides security verification facilities for applications. {@code
 * SecurityManager} contains a set of {@code checkXXX} methods which determine
 * if it is safe to perform a specific operation such as establishing network
 * connections, modifying files, and many more. In general, these methods simply
 * return if they allow the application to perform the operation; if an
 * operation is not allowed, then they throw a {@link SecurityException}. The
```

```
 * only exception is {@link #checkTopLevelWindow(Object)}, which returns a
 * boolean to indicate permission.
 */
public class SecurityManager {
…
    /**
     * Checks whether the calling thread is allowed to access the resource being
     * guarded by the specified permission object.
     *
     * @param permission
     *            the permission to check.
     * @throws SecurityException
     *             if the requested {@code permission} is denied according to
     *             the current security policy.
     */
    public void checkPermission(Permission permission) {
        try {
            inCheck = true;
            AccessController.checkPermission(permission);
        } finally {
            inCheck = false;
        }
    }

    /**
     * Checks whether the specified security context is allowed to access the
     * resource being guarded by the specified permission object.
     *
     * @param permission
     *            the permission to check.
     * @param context
     *            the security context for which to check permission.
     * @throws SecurityException
     *             if {@code context} is not an instance of {@code
     *             AccessControlContext} or if the requested {@code permission}
     *             is denied for {@code context} according to the current
     *             security policy.
     */
    public void checkPermission(Permission permission, Object context) {
        try {
            inCheck = true;
            // Must be an AccessControlContext. If we don't check
            // this, then applications could pass in an arbitrary
            // object which circumvents the security check.
            if (context instanceof AccessControlContext) {
                ((AccessControlContext) context).checkPermission(permission);
            } else {
                throw new SecurityException();
            }
        } finally {
            inCheck = false;
        }
    }
}
```

736. The SecurityManager achieves this result by calling the AccessController.checkPermission method, which in turn calls the AccessControlContext.checkPermission method to check the permissions. Further, the AccessControlContext method "checkPermission" checks each context to verify that each context has the appropriate permission, as shown here:

```
In Froyo, dalvik\libcore\security-kernel\src\main\java\java\security\AccessController.java
```

In Gingerbread, libcore\luni\src\main\java\java\security\AccessController.java:

```
    /**
     * Checks the specified permission against the vm's current security policy.
     * The check is performed in the context of the current thread. This method
     * returns silently if the permission is granted, otherwise an {@code
     * AccessControlException} is thrown.
     * <p>
     * A permission is considered granted if every {@link ProtectionDomain} in
     * the current execution context has been granted the specified permission.
     * If privileged operations are on the execution context, only the {@code
     * ProtectionDomain}s from the last privileged operation are taken into
     * account.
     * <p>
     * This method delegates the permission check to
     * {@link AccessControlContext#checkPermission(Permission)} on the current
     * callers' context obtained by {@link #getContext()}.
     *
     * @param permission
     *            the permission to check against the policy
     * @throws AccessControlException
     *             if the specified permission is not granted
     * @throws NullPointerException
     *             if the specified permission is {@code null}
     * @see AccessControlContext#checkPermission(Permission)
     *
     * @since Android 1.0
     */
    public static void checkPermission(Permission permission)
            throws AccessControlException {
        if (permission == null) {
            throw new NullPointerException("permission == null");
        }

        getContext().checkPermission(permission);
    }
```

In Froyo, dalvik\libcore\security-kernel\src\main\java\java\security\AccessControlContext.java
In Gingerbread, libcore\luni\src\main\java\java\security\AccessControlContext.java:

```
// List of ProtectionDomains wrapped by the AccessControlContext
// It has the following characteristics:
//    - 'context' can not be null
//    - never contains null(s)
//    - all elements are unique (no dups)
ProtectionDomain[] context;

...
    /**
     * Checks the specified permission against the vm's current security policy.
     * The check is based on this {@code AccessControlContext} as opposed to the
     * {@link AccessController#checkPermission(Permission)} method which
     * performs access checks based on the context of the current thread. This
     * method returns silently if the permission is granted, otherwise an
     * {@code AccessControlException} is thrown.
     * <p>
     * A permission is considered granted if every {@link ProtectionDomain} in
     * this context has been granted the specified permission.
     * <p>
     * If privileged operations are on the call stack, only the {@code
     * ProtectionDomain}s from the last privileged operation are taken into
     * account.
     * <p>
     * If inherited methods are on the call stack, the protection domains of the
     * declaring classes are checked, not the protection domains of the classes
     * on which the method is invoked.
     *
     * @param perm
     *            the permission to check against the policy
     * @throws AccessControlException
```

```
     *             if the specified permission is not granted
     * @throws NullPointerException
     *             if the specified permission is {@code null}
     * @see AccessController#checkPermission(Permission)
     */
    public void checkPermission(Permission perm) throws AccessControlException {
        if (perm == null) {
            throw new NullPointerException("Permission cannot be null");
        }
        for (int i = 0; i < context.length; i++) {
            if (!context[i].implies(perm)) {
                throw new AccessControlException("Permission check failed "
                        + perm, perm);
            }
        }
        if (inherited != null) {
            inherited.checkPermission(perm);
        }
    }
```

737. The AccessControlContext "checks the specified permission against the VM's current security policy," which is detailed in PolicyEntry.java. (*See* comment before checkPermission in AccessController.java.)

```
PolicyEntry.java:
In Froyo:
    /**
     * Checks if passed CodeSource matches this PolicyEntry. Null CodeSource of
     * PolicyEntry implies any CodeSource; non-null CodeSource forwards to its
     * imply() method.
     */
    public boolean impliesCodeSource(CodeSource codeSource) {
        return (cs == null) ? true : cs.implies(codeSource);
    }

    /**
     * Checks if specified Principals match this PolicyEntry. Null or empty set
     * of Principals of PolicyEntry implies any Principals; otherwise specified
     * array must contain all Principals of this PolicyEntry.
     */
    public boolean impliesPrincipals(Principal[] prs) {
        return PolicyUtils.matchSubset(principals, prs);
    }

    /**
     * Returns unmodifiable collection of permissions defined by this
     * PolicyEntry, may be <code>null</code>.
     */
    public Collection<Permission> getPermissions() {
        return permissions;
}

In Gingerbread:
    /**
     * Checks if passed CodeSource matches this PolicyEntry. Null CodeSource of
     * PolicyEntry implies any CodeSource; non-null CodeSource forwards to its
     * imply() method.
     */
    public boolean impliesCodeSource(CodeSource codeSource) {
        if (cs == null) {
            return true;
        }

        if (codeSource == null) {
            return false;
```

```
public void checkPermission(Permission perm) throws AccessControlException {
    if (perm == null) {
        throw new NullPointerException("Permission cannot be null");
    }
    for (int i = 0; i < context.length; i++) {
        if (!context[i].implies(perm)) {
            throw new AccessControlException("Permission check failed "
                    + perm, perm);
        }
    }
    if (inherited != null) {
        inherited.checkPermission(perm);
    }
}
```

763. The steps described above detail how Google's Android determines whether an action requested by a principal is permitted, based on permissions associated with a plurality of routines in a calling hierarchy, when one of the routines is privileged. The SecurityManager calls the AccessController, which in turn calls the AccessControlContext, which in turn calls the ProtectionDomain, which holds the permissions. The "checkPermission" method of the AccessControlContext checks only the ProtectionDomains after the last privileged operation if privileged operations are on the call stack. Therefore, Google's Android determines whether an action requested by a principal is permitted, based on permissions associated with a plurality of routines in a calling hierarchy, when one of the routines is privileged.

764. Therefore, in my opinion, Android literally meets the limitations of Claim 15 as recited.

765. Because the Android code meets the conditions of these claims, anyone having an binary version of Android, a system image on a device or provided with the Android SDK, has a tangible storage medium, *e.g.*, RAM, flash memory, or hard disk of a computer, storing code including instructions that, when executed by a processor, cause the processor perform the steps recited in Claim 13, 14, and 15. As stated in Oracle's infringement contentions submitted to Google on April 1, 2011, "Anyone who engages makes, uses, offers to sell, sells, or imports storage devices containing Android code directly infringes the computer-readable medium claims. This includes Google and its downstream licensees, including device manufacturers and application developers." The source code for these instructions is maintained by Google at its source repository for the Android source code. (*See, e.g.*,

http://source.android.com/source/downloading.html and http://android.git.kernel.org/.) These internet sites, on servers provided, operated, or maintained by Google, allow users to view and download the Android source code. These servers have hard disks, RAM, or other storage media carrying the sequences of instructions that make up the Android source code.

766. Therefore, in my opinion, Android literally meets the elements of Claims 13, 14, and 15 as recited.

XIII. CONCLUSION

767. For the foregoing reasons, it is my opinion that Android infringes:

- Claims 11, 12, 15, 17, 22, 27, 29, 38, 39, 40, and 41 of United States Patent No. RE38,104;
- Claims 1, 2, 3, and 8 of United States Patent No. 6,910,205;
- Claims 1, 6, 7, 12, 13, 15, and 16 of United States Patent No. 5,966,702;
- Claims 1, 4, 8, 12, 14, and 20 of United States Patent No. 6,061,520;
- Claims 1, 4, 6, 10, 13, 19, 21, and 22 of United States Patent No. 7,426,720;
- Claims 10 and 11 of United States Patent No. 6,125,447; and
- Claims 13, 14, and 15 of United States Patent No. 6,192,476

It is also my opinion that Google is liable for direct and indirect infringement in the manner described above.

768. For the forgoing reasons, it is my opinion that the patents-in-suit form the basis for consumer demand for Android by developers and end-users.

769. For the forgoing reasons, it is my opinion that once Google decided to adopt the Java execution model in Android, the patents-in-suit became necessary to Android achieving satisfactory performance and security.

Dated: August 8, 2011

John C. Mitchell